1. Computing with vectors & matrices

1.1. Numerics & Error Analysis

Real-world quantities: \( \mathbb{R} / \mathbb{Q} \)

Computers can't compute properly in \( \mathbb{R} \)

\( \Rightarrow \) Set of machine numbers \( M \): float & double

\( M \subseteq \mathbb{R} \)

\( M \) is not closed under basic arithmetic operations

\[ \text{op} \in \{ \ast, /, +, - \} \]

\[ \text{op}: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \]

\[ M \times M \rightarrow M \]

Code Snippet 1.1: Machine Arithmetic Example

```cpp
#include <iostream>

using namespace std;

double a = 1.0;
double b = a/9.0;
if (a == b + 9.0) cout << "They are equal";
else cout << "They are not equal";
```

will print: not equal

\( \frac{1}{9} = 0.1 \)

Instead of == queries: ask for approximate equality

some given tolerance
Code Snippet 1.2: Machine Arithmetic Example

```cpp
#include <limits>
#include <iostream>

using namespace std;

double a = 1.0;
double b = a/9.0;

if(fabs(a-b+9.0)<numeric_limits<double>::epsilon())
    cout << "They are equal";
else cout << "They are not equal";
/*
\text{introduce tradeoff between tolerance and accuracy}
\text{Fixed Point Representation:}
\text{fixed decimal point} \rightarrow \text{most straightforward way to store numbers}
\text{Range: } 10^{-k} \text{ to } 10^{-l}
\text{k, l \in N}
\text{k+l+1 digits \text{k} of which appear after decimal point}
\text{Default: Floating Point Representation}
\text{In applications useful with frequent change of scale} \rightarrow \text{requires unified representation}
\text{Pro: arithmetic operations: almost as if working with integers}
a+b = (a \cdot 10^k + b \cdot 10^k) \cdot 10^{-k}
\text{Cov: precision issue}
e.g. k=1: 0.1 \times 0.1 = 0.01 = 0 \text{ truncation}
\text{Used in systems that favor time over accuracy (e.g. GPU systems)}
\text{Pro: arithmetic operations: almost as if working with integers}
a+b = (a \cdot 10^k + b \cdot 10^k) \cdot 10^{-k}
\text{Cov: precision issue}
e.g. k=1: 0.1 \times 0.1 = 0.01 = 0 \text{ truncation}
\text{Used in systems that favor time over accuracy (e.g. GPU systems)}
\text{Pro: arithmetic operations: almost as if working with integers}
a+b = (a \cdot 10^k + b \cdot 10^k) \cdot 10^{-k}
\text{Cov: precision issue}
e.g. k=1: 0.1 \times 0.1 = 0.01 = 0 \text{ truncation}
\text{Used in systems that favor time over accuracy (e.g. GPU systems)}
```
Floating Point Representation:

\[
\pm 0.73125 \cdot 10^{12}
\]

digits of mantissa

Definition 1.1.1 (Machine numbers/floating point numbers). Given:

- Basis \( B \in \mathbb{N} \setminus \{1\} \)
- Exponent range \( \{e_{\text{min}}, \ldots, e_{\text{max}}\}, e_{\text{min}}, e_{\text{max}} \in \mathbb{Z}, e_{\text{min}} < e_{\text{max}} \)
- Number \( m \in \mathbb{N} \) of digits (for mantissa)

The corresponding set of machine numbers is:

\[
M := \{ d \cdot B^E : d = i \cdot B^{-m}, i = B^{-m-1}, \ldots, B^m - 1, E \in \{e_{\text{min}}, \ldots, e_{\text{max}}\} \}
\]

Remark. Machine numbers are not evenly spaced!

Gaps are bigger for large numbers:

\[
\begin{array}{cccccccccccccccc}
0 & 1 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 1 \\
\hline
1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\text{spacing } B^{e_{\text{min}}-1} & \text{spacing } B^{e_{\text{min}}+m-1} & \text{spacing } B^{e_{\text{max}}+m+2} \\
\text{Gap partly filled with non-normalized numbers}
\end{array}
\]

Recall:

\( \odot : M \times M \rightarrow M \)

On computer:

\( \tilde{\odot} : M \times M \rightarrow M \)

Implementation:

\( \tilde{\odot} = \text{rd} \circ \odot \)
Definition 1.1.4 (Correct rounding). Correct rounding ("rounding up") is given by the function

\[
\text{rd} : \begin{array}{c}
\mathbb{R} 
\rightarrow 
\mathbb{M} \\
x 
\mapsto \max_{x \in \mathbb{M}} \{x - \tilde{x}\} 
\end{array}
\]

Definition 1.1.2 (Absolute and relative error). Let \( \tilde{x} \in \mathbb{K} \) be an approximation of \( x \in \mathbb{K} \). Then its absolute error is given by

\[
\epsilon_{\text{abs}} := |x - \tilde{x}|,
\]

and its relative error is defined as

\[
\epsilon_{\text{rel}} := \frac{|x - \tilde{x}|}{|x|}.
\]

Approximation \( \tilde{x} \) of \( x \) has \( \epsilon_{\text{abs}} \) correct digits if \( \epsilon_{\text{rel}} \leq 10^{-\ell} \).

Maximal relative error of rounding:

\[
\epsilon_{\text{max}} = \max_{x \in \mathbb{R}} \frac{|\text{rd}(x) - x|}{1|x|},
\]

Assumption 1.1.1 ("Axiom" of roundoff analysis). There is a small positive number \( \epsilon_{\text{max}} \), the machine precision, such that, for the elementary arithmetic operations \( * \in \{+,-,\cdot,\} \) and "hard-wired" functions \( f \in \{\exp, \sin, \cos, \log, \ldots\} \), the following holds:

\[
x \tilde{x} y = (x \times y)(1 + \delta), \quad \tilde{f}(x) = f(x)(1 + \delta) \quad \forall x, y \in \mathbb{M},
\]

Alternative way to understand \( \epsilon_{\text{max}} \):

Smallest positive number s.t.

\[
1 \leq \epsilon_{\text{max}} + 1. \quad (\text{note: e.g.} 10 \approx \epsilon_{\text{max}} = 10)
\]
Note: other sources of errors exist:
  - Measurement error
  - Modeling error
  - Discretization error
  - etc.

forward error: \( \| x_{ex} - x_{app} \| \)

backward error: \( \| b - b_{app} \| \ll \) computable

In practice: stop when \( A x_{app} - b \) small

Note: Rel. & abs. error: in general not computable
  (don't know true solution!)

However:
  - \( x_{app} \) can be far off from \( x_{ex} \)

1. Worst case estimates

2. Compute backward error

Suppose we want to solve for

\[ Ax = b \quad x_{ex} \text{ (true sol.)} \]

compute \( x_{app} \)

\[ b_{app} = Ax_{app} \]

compare \( b_{app} \) to \( b \)

Condition number:

Example of matrix \( A \):

\[ \text{cond} (A) = \frac{\sigma_{\text{max}}}{\sigma_{\text{min}}} \]

\( \sigma_{\text{max}} \) ratio of largest & smallest singular value
Recall toy example: 3x3 matrix

\[ \|b - b^*\| = 0.007 \quad \|x - x^*\| \text{ huge} \]

condition number of \( A \): \( 10^5 \)

1.2 Fundamentals

1.2.1. Notation

\[ A := \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \in \mathbb{K}^{m \times n} \]

\( A^T \) transpose of \( A \)

\( A^H \) adjoint of \( A \)

\[ (A)_{ij} = a_{ij} \quad i \in \{1, \ldots, m\} \]

\[ j \in \{1, \ldots, n\} \]

\[ (A)_{ij} = (A)_{ji} \quad i \neq j \]

\[ (A)_{ij} = (A)_{ji} \quad i = j \]

\[ A = \mathbb{R}^{m \times n} : A^H = A^T \]

\[ A = \mathbb{C}^{m \times n} : A^H = \begin{bmatrix} \overline{a_{11}} & \cdots & \overline{a_{1n}} \\ \vdots & \ddots & \vdots \\ \overline{a_{m1}} & \cdots & \overline{a_{mn}} \end{bmatrix} \]

\[ A \text{ symmetric} : A^T = A, \quad A \text{ Hermitian} : A^H = A \]
Definition (s.p.d. matrix)

The matrix $A \in \mathbb{K}^{n \times n}$, where $n \in \mathbb{N}$, is symmetric (or Hermitian) (semi-)positive definite (s.p.d.) if

$$A = A^H$$

and

$$\forall x \in \mathbb{K}^n : x^H A x \geq 0$$

and

$$x^H A x > 0 \iff x \neq 0$$

(≥)

1.3. Libraries

1.3.1 Eigen

Header-only C++ library for numerical computations

- Provides data structures
- (Standard) operations

- (Any) matrices/vectors

- Fundamental data type: matrix

A matrix $A \in \mathbb{K}^{n \times n}$ is symmetric (semi-) positive definite if and only if all its eigenvalues are positive (non-negative).

Matrix<

typename scalar, int rows, int cols>

Typo of coeffs

(double, long, ...)

Fixed / Dynamic
Example: \texttt{Matrix<double, Dynamic, Dynamic>}

- size not known at compile time but
  treated as runtime variable

\texttt{Convenience typedefs:}

\texttt{MatrixXd} \<\texttt{double}\> \<\texttt{3x3 matrix}\>

\texttt{Matrix3f} \<\texttt{float}\> \<\texttt{3x3 matrix}\>

\texttt{MatrixXd} \<\texttt{3x3 matrix with array of uninitialized coeffs}\>

\texttt{MatrixXf} \<\texttt{Dynamic size, current size 0-by-0}\>

\texttt{MatrixXf} \texttt{y(6,9);}

\texttt{VectorXd} \texttt{x(12);}

\texttt{x.resize(5);}

\texttt{dynamic size}

\texttt{array of coeffs, allocated with given size}

\texttt{Code Snippet 1.9: Initializing special matrices in Eigen}

\texttt{#include \texttt{<Eigen/Dense>}}

\texttt{// Just allocate space for matrix, no initialisation}
\texttt{Eigen::MatrixXd A(rows, cols);}  
\texttt{// Zero matrix. Similar to matlab command zeros(rows, cols);}  
\texttt{Eigen::MatrixXd B = MatrixXd::Zero(rows, cols);}  
\texttt{// Ones matrix. Similar to matlab command ones(rows, cols);}  
\texttt{Eigen::MatrixXd C = MatrixXd::Ones(rows, cols);}  
\texttt{// Matrix with all entries same as value.}  
\texttt{Eigen::MatrixXd D = MatrixXd::Constant(rows, cols, value);}  
\texttt{// Random matrix, entries uniformly distributed in [0,1]}  
\texttt{Eigen::MatrixXd E = MatrixXd::Random(rows, cols);}  
\texttt{// (Generalized) identity matrix, 1 on main diagonal}  
\texttt{Eigen::MatrixXd I = MatrixXd::Identity(rows, cols);}  
\texttt{std::cout << \"size of A = \(\) \texttt{\(A.rows()\)}, \texttt{\(A.cols()\)} \texttt{\(=\) \texttt{\(0\)}};\texttt{std::endl;}
1.3.2. Dense Matrix Storage Formats

A ∈ \mathbb{R}^{m \times n}: stored as array of length m\cdot n

\[ A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 4 & 5 & 6 & 7 & 8 & 9 & 1 & 2 & 3 \end{pmatrix} \]

- **Row major** (C-arrays, bitmaps, Python):
  - A \_arr 1 2 3 4 5 6 7 8 9

- **Column major** (Fortran, MATLAB, Eigen):
  - A \_arr 1 4 7 2 5 8 3 6 9

Indices start at 0!

To access coefficients \( A(i, j) \):

\[ A(4) = 5 \]

Default: column major (in Eigen)

\[ \Rightarrow \text{can be changed} \]

---

// Template parameter ColMajor selects column major data layout
Matrix<\texttt{double},\texttt{Dynamic},\texttt{Dynamic},\texttt{ColMajor}> mm(nrows, ncols);

// Template parameter RowMajor selects row major data layout
Matrix<\texttt{double},\texttt{Dynamic},\texttt{Dynamic},\texttt{RowMajor}> mm(nrows, ncols);

Data storage format impacts runtime

Example: row/column access of a matrix

- Stored in column major

- Significantly more runtime as matrix size increases

---

1.4. Computational Effort

Computational effort \( \hat{=} \) number of elementary operations in a run
1.4.1. Asymptotic complexity

How does the computational effort scale with the problem size?

→ comparison of algorithms & their performance

**Definition 1.5.1 (Asymptotic complexity).** The asymptotic complexity of an algorithm characterises the worst-case dependence of its computational effort on one or more problem size parameter(s) when these tend to $\infty$.

**Typical parameter:** dimension of input vector, matrix

**Worst case analysis:** What is the maximal effort over the set of all admissible inputs?

Landau $\Theta$-notation:

$$f(n) = \Theta(g(n)) \quad f, g : \mathbb{N} \to \mathbb{R}$$

if \( \exists C > 0 \) \( \forall n \in \mathbb{N} \) s.t.

\[ \forall n \geq n^* : f(n) \leq C \cdot g(n). \]

Asymptotic complexity predicts independence of runtime on the size of the problem.

**Ex.:** \( \text{cost}(n) = \Theta(n^2) \)

**Example:** Conjecture \( t_i \propto n_i^k \)

\[ \text{runtimes} \quad \text{problem sizes} \]

\[ i = 1, \ldots, N \]
log-log-plot: \( \log t_i \approx \log C + \alpha \log n_i \)

data points \((t_i, n_i)\) lie on a straight line with slope \(\alpha\).

1.4.2. Cost of basic operations